

PREFACE

This paper presents a comparison of C-based object systems. To provide a basis for that comparison, the articles begin with a perspective about why OOP is important, what the important issues are with object orientation, and a brief mention of other object systems from which designers of C-based object systems have drawn their inspiration.

SOFTWARE WITH CLASS?

Programming practices have evolved with one primary motivation: improve the ability to maintain and reuse software. Object-oriented programming came about in an attempt to make code more modular and easier to reuse. The ideas behind object orientation are simple. To make code more modular, combine data structures and functions together: create software modules that are inherently self-contained. In other words, prefabricate software so that code functionality comes in one piece. Each module becomes a class.

To improve the ability to reuse software, allow classes to inherit from one another. This way, a new class can get all the functionality it needs from other classes, with the exception of the behavior that makes the new class unique. So, the two big concepts behind object orientation are 1) modular software components (by combining data structure and behavior), and 2) inheritance. While modularity, achieved through the use of classes, simplifies code maintenance, inheritance provides the sizzle of easy reuse by letting a programmer incrementally modify and expand class behavior.

The continuing challenge of software development is managing complexity. Object-oriented programming, with its inherent high degree of modularity, helps. But class libraries by no means ensure that software complexity is well managed. Quite the contrary, there is a disturbing trend in object-oriented software construction towards class libraries with hundreds of classes, but without the integration between them to simplify usage or maximize productivity. It may well turn out that the OOP development tools that stand the test of time are not those that try to offer everything through diversity, but those that integrate well the basics that most applications need. But class library architecture is the subject another paper. This one is about object systems.

OBJECT SYSTEMS

Comparing C-Based Object Systems

Gary Odom, Electron Mining, emine@aol.com

Abstract:

An overview of the important issues of object-oriented programming, and a comparison of C-based object systems (Objective-C, C++, and OOPC).

Object orientation is implemented by an object system. An object system is the way in which object-oriented programming is done, determining which features of object orientation are supported, and to what extent. Most often, an object system is built into a new language, or becomes an extension to an existing procedural programming language. Object-oriented language extensions often require a new compiler, though C-based object systems are usually implemented using a preprocessor to a C compiler. The disadvantage to using such extensive preprocessing is that providing source-level debugging is difficult.

The success of any programming language is the ability to give a programmer range and flexibility in implementing software designs in the most straight-forward manner. C usurped FORTRAN and Pascal in popularity largely because C allows a programmer greater range (with such features as permitting a variable number of function arguments, and built-in bit twiddling), and flexibility in expression.

There has been heightened interest in recent years with visual programming environments. Just as creative people have historically chosen between literary or graphic artistic expression, perhaps we are beginning an age where software developers will have a similar choice in their medium. While this article is focused on written OOP languages, the same evaluation criteria may be applied to visual OOP tools.

Inheritance

Because inheritance is one of the key concepts behind object orientation, one way to judge the quality of an object system is how flexibly inheritance can be specified. Multiple inheritance is the ability of a class to inherit from multiple classes. With multiple inheritance, a new set of methods (behavioral functions) can easily be added to an existing class. So, for example, you could attach a debugging class to another class without introducing new sequential links in the inheritance chain; the debugging class could verify the validity of data in objects of the target class. Most often, multiple inheritance is used to create a subclass that combines a class with primary functionality with another class that adds some ancillary characteristics. For example, a text graphic class (`cTextGraphic`) would inherit from a text class (`cText`) for text processing, plus inherit from a graphic class (`cGraphic`) to allow a user to treat the text as a graphic object (as in an object-oriented drawing program). Because it is so convenient, most current object systems support multiple inheritance.

An advanced object system allows inheritance and methods to be defined dynamically, while a program runs. This is called dynamic definition.

Class-Object Schizophrenia

The theory of object orientation makes a clear distinction between classes and objects. Classes are object factories, templates that exist only in source code. A class specifies object data structure, while a class itself has no data. A class just has methods, so that objects can take function calls. Objects alone exist as dynamic entities in memory as a program runs. An object can't have methods separate from the class it inherits from, and a class can't have its own data.

This fundamental distinction between classes and objects can be blurred to considerable benefit. Smalltalk and Objective-C allow classes to have their own methods (class methods), apart from the methods an object that inherits from the class has.

In an advanced object system using flexible class-object construction, classes may have their own data structure (class variables) and their own methods (class methods), and objects may have their own methods (object methods), separate from the class methods they inherit. These capabilities provide

flexibility in software design and implementation, as well as giving conceptual consistency to working with objects. While theory may put a wall between classes and objects, eliminating class-object distinctions gives a developer great practical flexibility in meeting design requirements.

There is another aspect of object orientation that defines the quality of an object system: dispatch control.

Dispatch Control

Object orientation introduces a rather strange concept: calling a function without knowing exactly what function is going to be called. This happens because different classes can use the same function name. For example, to draw an object, you might write `draw(self)`, where `self` is the object to be drawn. A dispatch mechanism is used to find the right method to call based upon the class inheritance of the `self` object. The technical term for this function-calling shell game is polymorphism (Greek (to me) for "multiple shapes"). Polymorphism is great because it lets code become very general: you can draw all objects on a page by calling `draw(self)` in a loop, where the loop assigns `self` from a page object array.

Polymorphism can mean finding the right class method to dispatch to at run time (dynamic binding), rather than binding a function call at compile time (static binding) (what linkers do for a living). Method dispatch with dynamic binding is one overhead imposed by object orientation. This overhead is the price paid for quicker development time, smaller code size, flexibility in using prefabricated software, and easier maintenance. Hybrid languages, such as C++, let a programmer go back to procedural programming for time-critical code, whereas this is not an option with a pure object-oriented language such as Smalltalk.

Comparing C-Based Object Systems

Gary Odom, Electron Mining, emine@aol.com

Abstract:

An overview of the important issues of object-oriented programming, and a comparison of C-based object systems (Objective-C, C++, and OOPC).

Comparing C-Based Object Systems

Just as flexibility in specifying inheritance is important, so is flexibility in dispatch. Features of dispatch flexibility are being able to call multiple methods by a single function call (multiple dispatch), controlling which methods are called and in what order (dispatch control), being able to dispatch to a specific method, and dispatching based upon multiple arguments (called multi-methods).

The essence of high-quality dispatch in an object system is being able to call multiple methods in a single function call (multiple dispatch), and being able to control method call order (dispatch control). Imagine a resource-based picture class (`cPicture`), which inherits from a resource class (`cResource`). To draw a `cPicture` object (`draw(picture)`), you want to first make sure the picture resource is in memory. The `cResource` is used as a before-method, to check and load the resource if it has been purged. The `cPicture` `draw` method, which draws the picture, is an after-method.

An advanced object system allows dispatch control using before- and after-methods. A truly flexible object system lets dispatch control be altered dynamically, while a program runs, as part of dynamic definition.

Another aspect of dispatch control is being able to dispatch to a specific class method, rather than accepting the default dispatch. For example, you may want to draw just the handles on a graphic object by calling `dispatch_to(cGraphic, draw, self)`, rather than calling `draw(self)`, which draws an object and its handles. Almost all object systems offer this capability.

Multi-methods are methods dispatched based upon multiple arguments.

Object Links

One of the problems with procedural programming is that it takes effort to build self-contained, reusable software modules. But it is easy to link data structures through functions.

In a role reversal to procedural programming, the modular, decentralized nature of object orientation presents an interesting design decision: how best to link and integrate related objects (and classes). A significant challenge with object-oriented programming is providing systematic links

between objects of different classes. While object links are the basis for object-oriented databases (OODB), they are also a necessary ingredient of any object-oriented application. Garbage collection can be facilitated using object links. Object links can be done willy-nilly using pointers in object data, but such an approach isn't ideal for use in garbage collection, or OODB construction. Because object links are a structural element of any object-oriented application, a good object system should offer built-in support for object links.

Dynamic Definition

The single most important feature of an object system is its level of dynamism. A fully dynamic object system allows inheritance and methods to be defined, and redefined, while an application is running. Dynamic definition permits great flexibility in software construction. There is a wide chasm in object-oriented power between static and dynamic object systems.

A simple example of dynamic inheritance : reading in dialog item (DITL) resources from a file, creating dialog item objects, then adding the right class (control, picture, text, etc.) to a dialog item once the item type is discovered (by reading the resource data). This can be done in a static language by not assigning the dialog item type class before reading the dialog item resource definition, but that involves processing in a way dictated by the language's limitations, rather than doing things the way that might first come to mind (which is usually the easiest way) if no constraints were imposed. The flexibility of a dynamic object system brings both small and large benefits.

Comparing C-Based Object Systems

Gary Odom, Electron Mining, emine@aol.com

Abstract:

An overview of the important issues of object-oriented programming, and a comparison of C-based object systems (Objective-C, C++, and OOPC).

OBJECT SYSTEM SURVEY

Smalltalk

Dating back to 1967, Simula was the first object-oriented language. But, because of its looming influence, Smalltalk is the grandmother of object-oriented programming languages. Smalltalk was designed as part of an object-oriented environment, with hundreds of classes, where everything is object-oriented. There is no class-object distinction with Smalltalk. Using the Smalltalk environment is a “deep immersion” experience in a land of objects, which is why it has been such an inspiration.

Smalltalk has a surprising limitation: it does not support multiple inheritance. Because even the simplest message uses dynamic binding (even the $+ \text{ in } C = A + B$), Smalltalk is slow.

The phraseology of “sending messages to objects” is a holdover from Smalltalk, where the syntax is object-verb (such as `thisOval draw` to draw `thisOval`), rather than the more typical function-calling paradigm of verb-object (such as `draw(thisOval)`). As with most languages, the verb-object function call model is used in this article.

CLOS

The Common Lisp Object System, known as CLOS, is the ANSI-standard language extension to Lisp that adds object orientation. CLOS is noteworthy because, in a sea of tug-boat object systems, CLOS is a luxury liner. CLOS supports multiple inheritance, dispatch control, multi-methods, flexible class-object construction, dynamic binding and dynamic definition. (CLOS has a dynamic object system.) To simplify the application programming interface (API), CLOS consistently uses generic functions. Generic functions are polymorphic functions, such as `draw` and `act`. The object orientation that CLOS allows is tremendously flexible and expressive, but because Lisp has a limited domain, namely AI and list/language processing, CLOS, like Lisp, will never become a mainstream language.

Apple's new Dylan language is an ambitious version of CLOS (ambitious in its kitchen-sink feature set; sort of “CLOS with an ADA mindset”).

C OBJECT SYSTEMS

Because of its simplicity, flexibility, efficiency and range, C has become the industry choice for systems and application software development. It is natural to extend C into the object-oriented realm. A few interesting attempts have been made.

Objective-C

An early attempt to make C object oriented was Objective-C. Objective-C adds Smalltalk-like object orientation using a strict superset of C. Objective-C adds a class definition mechanism, an object data type, and a message expression type. In Objective-C, each class is defined by two files: an interface file, and an implementation file. The interface file specifies the class programming interface: class and superclass names, along with instance variable (object) declarations and method declarations. The implementation file has class method code.

Objective-C supports multiple inheritance. Like Smalltalk, Objective-C permits class methods. Like C++, Objective-C provides ways to enforce data hiding and restrict method access. Objective-C lacks dispatch control or dynamic definition.

Included in the NeXT operating system environment is a set of classes written in Objective-C for application development. Because these classes are native to the platform, NeXT application development is relatively easy, especially compared to the complex nightmare of the Macintosh Toolbox. Though there is little marketing of the product, Objective-C is available on the Macintosh as an MPW C preprocessor.

Commercially, Objective-C was ahead of its time. Its corporate sponsor, Stepstone Corporation, was near financial death before being resuscitated by adoption for the NeXT line of workstations. Now that NeXT itself has one foot in the grave (having given up making hardware after a flood of red ink), the long-term prospects of Objective-C are once again under a cloud.

Comparing C-Based Object Systems

Gary Odom, Electron Mining, emine@aol.com

Abstract:

An overview of the important issues of object-oriented programming, and a comparison of C-based object systems (Objective-C, C++, and OOPC).

C++

C++ is another language extension to C. Only part of the C++ extensions have to do with object orientation. Operator overloading, for example, adds flexibility to C, but has nothing to do with object orientation per se (although the use of overloading in C++ is restricted to the object oriented aspects of C++).

The object-oriented part of C++ implements a limited version of object orientation. Multiple inheritance is supported, as are class variables, class methods, multi-methods and optional dynamic binding, but C++ lacks dispatch control or dynamic definition. C++ classes have automatic initialization and deallocation methods.

C++ class constructs provide three levels of enforced information hiding. Access to data or methods can be private, protected or public. Restrictions can be overridden (by friend classes). The information-hiding features require new language syntax that complicates what was (in C) a lean language definition. Further, this feature sits in odd contrast to C's celebrated openness with typecasting, data manipulation and the free use of function pointers.

OOPC

OOPC (pronounced "oop-sea") is an acronym for "Object Oriented Programming in C". OOPC has an unusual implementation, in that it is not an extension to the C language, but rather a set of functions that turns C into an object-oriented language. Object-oriented programs written in OOPC look like standard C code, because they are just that. This consistency with C simplifies learning and using OOPC.

The look and feel of OOPC, while simple, is deceiving. OOPC has all the features of CLOS: multiple inheritance, dispatch control, flexible class-object construction, and dynamic definition. Multi-methods can be simulated. Plus, OOPC comes with built-in support for object links.

Like C++, OOPC provides automatic initialization and deallocation methods. OOPC also implements a form of garbage collection to prevent an object from being released

while it is still linked to any other object. Unlike Objective-C or C++, OOPC does not enforce data hiding or restrict method access.

OOPC always uses dynamic binding, but this overhead is minimized by using a dispatch table, which essentially results in static binding while still allowing dispatch control options.

To simplify the programming interface, OOPC consistently uses verb functions. OOPC verb functions are the same as CLOS generic functions: verbs used as polymorphic functions.

Although it has existed since 1988, OOPC has only been released to the public as a commercial product since 1992. The OOPC object system is only part of the product currently sold for Macintosh application development. OOPC includes a set of code libraries for rapid application development. Some low-level function libraries exist for efficiency and interface to the native operating system. OOPC also comes with a class library that provide automatic document management, an application user interface, a graphics package, and sophisticated, multiple-priority, threaded event handling. The consistent use of verb functions, streamlined class architecture and integration between classes simplify learning and using the OOPC class library, thus making OOPC suitable for novice and professional programmer alike.

Table 1 compares the features of the three C-based object systems discussed.

Table 1. Comparison of C-Based Object Systems

Feature	Obj-C	C++	OOPC
Multiple Inheritance	Yes	Yes	Yes
Class Variables	No	Yes	Yes
Class Methods	Yes	Yes	Yes
Multi-Methods	No	No	Yes
Object Methods	No	No	Yes
Multiple Dispatch	No	No	Yes
Dispatch Control	No	No	Yes
OODB Support	No	No	Yes
Dynamic Binding	Yes	Yes	Yes
Dynamic Definition	No	No	Yes

Comparing C-Based Object Systems

Gary Odom, Electron Mining, emine@aol.com

Abstract:

An overview of the important issues of object-oriented programming, and a comparison of C-based object systems (Objective-C, C++, and OOPC).